بسم الله الرحمن الرحيم

Introduction to operator overloading

مقدمة :

اعادة تعريف المعاملات او التحميل الزائد مع اني اؤيد اعادة التعريف ودائما نقع في مشكلة تعريب المصطلحات .

Operator overloading يسمح للمستخدم بتعريف كيفية عمل العمليات مثل (+ . - .++ . = ...) مع مختلف انواع البيانات .

وجميع العمليات بالسي ++ معرفة مثل عملية الجمع معرفة على انها تقوم بجمع العنصر الاول مع الثاني وارجاع الناتج

مثال

```
int nX = 2;
int nY = 3;
cout << nX + nY << endl;
```

راح يجمع nX and nY ثم يرجع القيمة ويطبعها على الشاشة ..لكن خلونا نشوف المثال التالي :

```
Mystring cString1 = "Hello, ";
Mystring cString2 = "World!";
cout << cString1 + cString2 << endl;
```

وشى تظن انة راح يطبع Hello, Word! !!!!!!!

طبعا خطاء لان + (عملية الجمع) لم نخبرها ماذا تعمل مع الكلاس الخاص بنا الي هو Mystring

في المثال السابق عند جمع العددين من نوع int ال+ (عملية الجمع) معرفة ضمن هذا الكلاس الي هو int

يجب علينا ان نخبر + (عملية الجمع) مع Mystring ماذا تعمل حتى نحصل على نتيجة

طيب لو اخبرنا + مع ال Mystringماذا تعمل .

وكتبنا برنامج يوجد فية جمع عددين من نوع int

ودمج 2 stringمن نوع Mystring ...

السؤال هل يتغير تعريف ال+ كليا بحيث ان اذا عرفناها ماذا تعمل مع Mystring لا يمكنها العمل مع ال int؟؟

طبعا ..اكيد لا على حسب البارمتر المرسل لها اذا اضفت 2 من نوع Mystring

فسوف تعمل على تعريف + الخاص بي Mystring

واذا اضفت 2 من نوع int فسوف تعمل على تعريف + الخاص بي int المعرف مسبقا ضمن السي ++

وهذا جدول يحوي على جميع العمليات التي يجوز اعادة تعريفها

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

هنا بعض النقاط الي لازم تحطها براسك :

- عند تعريف كلاس جديد وتريد تعريف + (عملية الجمع) فية فان + (عملية الجمع) محصورة ضمن نطاق الكلاس هذا ولا تستطيع تعريف + لي تعمل مع كلاسين يعني انك ما تستطيع انك تعيد تعريف + لتعمل مع واحد int وواحد float.

- تستطيع اعادة تعريف العمليات الموجودة بالجدول فقط ولا تستطيع انشاء عمليات جديدة .

وهذي مقدمة في الاوفر لودينك اعادة تعريف المعاملات .

2-Overloading the arithmetic operators

اكثر الاعمليات الحسابية استخدم هي + ،- ، * ، / جميع هذي العمليات هي عمليات ثنائية اي تستخدم عنصرين لأجراء العملية .

A-Overloading operators using friend functions

اذا كنا ما نريد اجراء تغير على المعاملات (اعضاء الكلاس) فأفضل طريق ان نستخدم الفريند فنكشن .

خلونا نعيد تعريف عملية + .لتقوم بعملية جمع كلاسين من تصميمنا نشوف المثال

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Add Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

int main()
{
    Cents cCents1(6);
    Cents cCents2(8);
    Cents cCentsSum = cCents1 + cCents2;
    std::cout << "I have " << cCentsSum .GetCents() << " cents." <<
std::endl;

    return 0;
}
```

الناتج راح يكون :

```
I have 14 cents.
```

عند اعادة تعريف اي عملية نستخدم كلمة operator ثم بعدها العملية الحسابية .

```
friend Cents operator+(const Cents &c1, const Cents &c2);
```

هذي الطريقة التي يتم فيها تعريف اي عملية بواسطة فنكشن الفريند .هنا تستقبل وسطين من نفس نوع الكلاس .و ماراح نعدل على الوسطين جعلناهم const .

خلونا نشوف تعريف الدالة :

```
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}
```

العملية بسيطة جدا تقوم بأرجاع كلاس Cents مع بارمتير مجموع الاعضاء الخاصة لكل من الاوبجكت الاول والثاني .

ولذاك لان الدالة فريند فا بالامكان الوصول الى اعضاء الاوبجكت الخاص

حنى وشى نستفيد من عملية الجمع هذي ....!!!!!

أصلن الاوبجكت Cents ما يحتوي الا على عضو واحد خاص فااذا بغينا نجمع الاوبجكت مع اوبجكت ثاني نقوم بجمع اعضائهم الخاصة وهذي هي الفكرة .

نواصل شرح الكود :

```
Cents cCents1(6);
Cents cCents2(8);
Cents cCentsSum = cCents1 + cCents2;
```

بالبداية واضح تم تعريف 2 اوبجكت وارسال قيم ابتدائية الي هي 6 و 8

الفكرة كلها بالسطر الاخير هذا :

```
Cents cCentsSum = cCents1 + cCents2;
```

بما ان الدالة

```
Cents operator+
```

عضو بالاوبجكت

فانا نساطيع كتابتها بالشكل التالي لكي تتضح الامور

```
Cents cCentsSum =cCents1.operator+ (cCents2);
```

و قد عرفنا الدالة في السابق على الشكل التالي:

```
Cents operator+(const Cents &c1, const Cents &c2)
```

فا cCents1 يعتبر الوسيط الاول و cCents2 يعتبر الوسيط الثاني .

طيب خلونا نعيد تعريف ـــ السالب

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // overload Cents + Cents
    friend Cents operator+(const Cents &c1, const Cents &c2);

    // overload Cents - Cents
    friend Cents operator-(const Cents &c1, const Cents &c2);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator+(int, int)
    return Cents(c1.m_nCents + c2.m_nCents);
}

// note: this function is not a member function!
Cents operator-(const Cents &c1, const Cents &c2)
{
    // use the Cents constructor and operator-(int, int)
    return Cents(c1.m_nCents - c2.m_nCents);
}
```

أعتقد واضحة الامور ولا يوجد فرق سواء بالضرب او القسمة .

عدل على البرنامج السابق بحيث يقوم بالعمليات الاربع الجمع والطرح والقسمة والضرب ؟(اذا حليتة فا اموورك ماشية تمام)

B-**Overloading operators for operands of different types**

اذا اردنا ان العملية + تعمل مع اشكال مختلفة مثلا لو نرجع لمثالنا السابق Cents(4)

وبغينا نضيف 6 لهذا الكلاس والناتج راح يكون Cents(10) .

في السي ++ عملية الجمع بين عددين مثل X+Y ما تختلف عن Y+X لانهم نفس النوع عملية التبديل ما تهم .

نفس العملية تحدث في operator+ عند استدعائها .على العموم اذا كان المتغيرين ما هم من نفس النوع فهنا نجب ان نحذر .

مثلا 6 + Cents(4) راح تستدعي الدالة operator+(Cents, int) و 6 + Cents(4) راح تستدعي .operator+(int, Cents) طبعا اذا كانت الانواع المررة للدالة من نفس النوع ما تفرق لكن اذا كانت تختلف نحتاج الى كتابة دالتين .راح تتضح الصورة بالمثال ..:

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(const Cents &cCents, int nCents);

    // Overload int + cCents
    friend Cents operator+(int nCents, const Cents &cCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(const Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}

// note: this function is not a member function!
Cents operator+(int nCents, const Cents &cCents)
{
    return Cents(cCents.m_nCents + nCents);
}

int main()
{
    Cents c1 = Cents(4) + 6;
    Cents c2 = 6 + Cents(4);
    std::cout << "I have " << c1.GetCents() << " cents." << std::endl;
    std::cout << "I have " << c2.GetCents() << " cents." << std::endl;

    return 0;
}
```

لو تلاحظ عملية الاختلاف كانت في

```
Cents c1 = Cents(4) + 6;
Cents    c2 = 6 + Cents(4)
```

الترتيب فقط واحتجنا الى اعادة تعريفها .

نشووف مثال ثاني :-

```
class MinMax
{
private:
    int m_nMin; // The min value seen so far
    int m_nMax; // The max value seen so far

public:
    MinMax(int nMin, int nMax)
    {
        m_nMin = nMin;
        m_nMax = nMax;
    }

    int GetMin() { return m_nMin; }
    int GetMax() { return m_nMax; }

    friend MinMax operator+(const MinMax &cM1, const MinMax &cM2);
    friend MinMax operator+(const MinMax &cM, int nValue);
    friend MinMax operator+(int nValue, const MinMax &cM);
};

MinMax operator+(const MinMax &cM1, const MinMax &cM2)
{
    // Get the minimum value seen in cM1 and cM2
    int nMin = cM1.m_nMin < cM2.m_nMin ? cM1.m_nMin : cM2.m_nMin;

    // Get the maximum value seen in cM1 and cM2
    int nMax = cM1.m_nMax > cM2.m_nMax ? cM1.m_nMax : cM2.m_nMax;

    return MinMax(nMin, nMax);
}

MinMax operator+(const MinMax &cM, int nValue)
{
    // Get the minimum value seen in cM and nValue
    int nMin = cM.m_nMin < nValue ? cM.m_nMin : nValue;

    // Get the maximum value seen in cM and nValue
    int nMax = cM.m_nMax > nValue ? cM.m_nMax : nValue;

    return MinMax(nMin, nMax);
}

MinMax operator+(int nValue, const MinMax &cM)
{
    // call operator+(MinMax, nValue)
    return (cM + nValue);
}

int main()
{
    MinMax cM1(10, 15);
    MinMax cM2(8, 11);
    MinMax cM3(3, 12);

    MinMax cMFinal = cM1 + cM2 + 5 + 8 + cM3 + 16;

     std::cout << "Result: (" << cMFinal.GetMin() << ", " <<
         cMFinal.GetMax() << ")" << std::endl;

     return 0;
}
```

هذا البرنامج يقوم بجمع مجموعة من الكلاس واخراج اكبر قيمة واصغر قيمة لم تعتمد طريقة الجمع بالمعنى المفهوم لكن تمى اسغلالها بطريقة ذكية .

تم اعادة تعريف عملية الجمع لكي تستطيع التعامل مع

اوبجكت + اوبجكت

اوبجكت + عدد

عدد +اوبجكت

والناتج راح يكون

```
Result: (3, 16)
```

وبالتوفيق ...:)

3-Overloading the I/O operators

التحميل الزائد لأدوات الدخل والخرج :

>> and <<

اذا كان عندنا كلاس يحتوى على عدة اعضاء وحبينا نطبع الاعضاء على الشاشة مثلا نشوف الكلاس التالي

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};
```

اذا حبينا نطبع الاعضاء الموجدين داخل الكلاس نستخدم تقريبا الكود هذا

```
Point cPoint(5.0, 6.0, 7.0);
cout << "(" << cPoint.GetX() << ", " <<
    cPoint.GetY() << ", " <<
    cPoint.GetZ() << ")";
```

بس لو اعدنا تعريف المعامل >> و << لأستطعنا كتابتها بهذا الشكل

```
Point cPoint(5.0, 6.0, 7.0);
cout << cPoint;
```

راح يعطينا نفس النتائج وبكل سهولة .اعادة تعريف المعامل << نفس المعامل

operator+ (they are both binary operators)

لانة معامل ثنائي (<<) بس الاختلاف بالبارمتر parameter .

في الجملة هذي `cout << cPoint` يوجد معاملين المعامل الايسر الي هو cout اوبجكت والمعامل الايمن الي هوكلاس اوبجكت cPoint

Coutهي اوبجكت من الكلاس ostream .

تعريف overloaded function تقريبا بالشكل التالي

```
friend ostream& operator<< (ostream &out, Point &cPoint);
```

طبعا في السي ++ المعامل >> معرف للتعامل مع double لان في مثالنا السابق الاعضاء من نوع double

فقط الي علينا نعملة هو تعريف ال >> لطباعة Point.

خلونا نعيد تعريف المعامل في المثال السابق

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    friend ostream& operator<< (ostream &out, Point &cPoint);

    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can access
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}
```

ان شاء الله الكود واضح ومافية اي غموض فية ترك بسيط

نوعية القيمة المرجعة في

```
ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can access
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}
```

ماهو السبب في ارجاع out ولماذا تم استخدامة ..!!؟

طبعا السبب في ارجاع قيمة هو عند طباعة الكلاس وطباعة امر آخر معة بنفس اللحظة مثل

```
cout << cPoint << endl;
```

فهنا تحدث مشكلة اذا لم نقم بأراجاع قيمة .**فلو ان لم نقم بأراجاع قيمة (يعني void) عند وصل الكومبايلر الى**

```
cout << cPoint << endl;
```

فيتم معالجتها عن طريق نظام الاسبقية/ الترابط .

فتتم معالجتها بهذي الطريقة

```
(cout << cPoint )<< endl;
```

فحسب الاولوية الاقواس اولا

فتصبح كأننا كتبناها هكذا

```
cout << cPoint
```

فا بعد الاستدعا وتنفيذ الدالة لا ترجع شي هو يعني void

نرجع لتكملية السطر فيصبح عندنا

Void<<endl;

وهذي القيمة لا تعني شي وهو خطاء في الكمبايلر .

فهذا هو سبب ارجاع قيمة من نوع ostream

طيب لو قمنا بأرجاع قيمة من نوع ostream راح تم التنفيذ بالشكل التالي

بالبداية راح يتم التنفيذ

```
(cout << cPoint )<< endl;
```

يعني

```
cout << cPoint
```

ثم القيمة المرجعة هي cout

فيصبح التكملة لها هي

Cout<<endl;

وهذا التنفيذ الصحيح وهو سر ارجاع قيمة من نوع ostream

بشكل عام عند اعادة تعريف اي معامل ثنائي قيمة اليسرى لا بد ان ترجع شي ...*_^

الان نكمل الكود في المين

```
int main()
{
    Point cPoint1(2.0, 3.0, 4.0);
    Point cPoint2(6.0, 7.0, 8.0);

    using namespace std;
    cout << cPoint1 << " " << cPoint2 << endl;

    return 0;
}
```

المخرجات راح تكون بالشكل التالي

```
(2.0, 3.0, 4.0) (6.0, 7.0, 8.0)
```

Overloading >>

المعامل هذا مثلة مثل المعامل السابق .النقطة المختلفة في هذا المعامل ان cin هي اوبجكت من نوع istream فقط ...:)

و هذا مثال مع الكلاس point

```
class Point
{
private:
    double m_dX, m_dY, m_dZ;

public:
    Point(double dX=0.0, double dY=0.0, double dZ=0.0)
    {
    m_dX = dX;
    m_dY = dY;
    m_dZ = dZ;
    }

    friend ostream& operator<< (ostream &out, Point &cPoint);
    friend istream& operator>> (istream &in, Point &cPoint);
    double GetX() { return m_dX; }
    double GetY() { return m_dY; }
    double GetZ() { return m_dZ; }
};

ostream& operator<< (ostream &out, Point &cPoint)
{
    // Since operator<< is a friend of the Point class, we can access
    // Point's members directly.
    out << "(" << cPoint.m_dX << ", " <<
        cPoint.m_dY << ", " <<
        cPoint.m_dZ << ")";
    return out;
}

istream& operator>> (istream &in, Point &cPoint)
{
    in >> cPoint.m_dX;
    in >> cPoint.m_dY;
    in >> cPoint.m_dZ;
    return in;
}
```

و هذا مثال بسيط يستخدم العمليات << >>

```
int main()
{
    using namespace std;
    cout << "Enter a point: " << endl;

    Point cPoint;
    cin >> cPoint;

    cout << "You entered: " << cPoint << endl;

    return 0;
}
```

ولنفرض انك ادخلة

```
3.0 4.5 7.26
```

المخرجات راح تكون بالشكل التالي

```
You entered: (3, 4.5, 7.26)
```

وفي الاخير ما بقي لنا الا نقطة واحدة الي هي يفضل عند ارسال المتغير الثاني ان يرسل كا ثابت مثلا كنا نكتبة في الامثلة الماضية مثل كذا

```
friend ostream& operator<< (ostream &out, Point
&cPoint);
```

والافضل ان يرسل هكذا

```
friend ostream& operator<< (ostream &out, const
Point &cPoint);
```

هذا والله اعلم .

4-Overloading operators using member functions

اذا العملية لا تجري تعديل على احد الاعضاء فأضل طريقة لكتابة الفنكشن هي بأاستخدام الفريند فنكشن

friend function

كما فعلنا سابقا اما اذا حبينا نجري تعديل على احد الاعضاء فأضل طريقة هي بأاستخدام

Member function

>استخدام دالة العضو اسهل من استخدام الفريند فنكشن

وهذي بعض النقاط المهمة :

-الوسيط الايسر لابد ان يكون اوبجكت من نفس نوع الكلاس.

-الوسيط الايسر يأتي مضمنا this*

-اذا لم يكن الوسيط الايسر من نفس نوع الاوبجكت مثل

operator+(int, YourClass), or operator<<(ostream&, YourClass)

في هذي الحالة لابد من استخدام friend function

-عملية الاسناد (=) والاقواس ([])والاستدعاء () و-> يجب ان تستخدم member function

طيب ندخل بالامثلة وان شاء الله تتضح الامور:

#اعادة تعريف المعامل الاحادي (-)....بالطريقتين كا friend function

و member function

اول شي friend function

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload -cCents
    friend Cents operator-(const Cents &cCents);
};

// note: this function is not a member function!
Cents operator-(const Cents &cCents)
{
    return Cents(-cCents.m_nCents);
}
```

وعند تعريفها كا member function

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload -cCents
    Cents operator-();
};

// note: this function is a member function!
Cents Cents::operator-()
{
    return Cents(-m_nCents);
}
```

نشوف الان اوجة الاختلاف بين هذان الكودان لو نلاحظ بأستخدام ال member function لم نرسل اي بارمتر كيف تتم العملية ؟ .كما نعرف ان member function تتضمن *this وتؤشر الى نفس الاوبجكت الي من نفس الكلاس اكيد فدالة العضو تعمل علية .

وال friend functionلا تحتوي على this * فتحتاج الى بارمتر .

تذكر جيدا عندما يرى الكومبايلر prototype تبع الفنكشن

مثل هذا

```
Cents Cents::operator-();
```

فـأنـة يـقـوم بـتـحـويـلـهـا الـى

```
Cents operator-(const Cents *this)
```

وعند تعريف friend function (البروتايب prototype) يتم في الشكل التالي

```
Cents operator-(const Cents &cCents)
```

Overloading the binary addition (+) operator

سنقوم بأعادة تعريف المعالم بطريقتين طريقة friend function و member function.

الطريقة الاولى طريقة friend function

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    friend Cents operator+(Cents &cCents, int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is not a member function!
Cents operator+(Cents &cCents, int nCents)
{
    return Cents(cCents.m_nCents + nCents);
}
```

والطريقة الثانية بأستخدام member function

```
class Cents
{
private:
    int m_nCents;

public:
    Cents(int nCents) { m_nCents = nCents; }

    // Overload cCents + int
    Cents operator+(int nCents);

    int GetCents() { return m_nCents; }
};

// note: this function is a member function!
Cents Cents::operator+(int nCents)
{
    return Cents(m_nCents + nCents);
}
```

زي ما تلاحظوا في ال friend function تستقبل Parameter 2 وبالمقابل member function تستقبل parameter 1

بسبب ما شرحنا سابقنا وهو ان cCents في member function تتضمن *this .

معظم المبرمجين يفضلوا friend function لانها اوضح بالقراءة

وفي بعض الأحيان لابد من استخدامها لكي تقوم بعمل لا تستطيع ال member function القيام بة مثل

```
friend operator+(int, cCents)
```

هذي لا يمكن كتابتها بي member function لان العضو الي في اليسار ليس عضو في الكلاس

هذا والله أعلم ...

5-Overloading the increment and decrement operators

اعادة تعريف المعامل **increment (++) و (--) decrement** في هي
تحتوي على نوعين **postfix** (**eg.** `nX++; nY--;`) والنوع الاخر **prefix**

(**eg.** `++nX; --nY;`) .

نـبـد أ مـع

Overloading prefix increment and decrement

Prefix يتم إعادة تعريفها مثل المعاملات الأحادية **unary**

خلونا نشوف المثال

```
class Digit
{
private:
    int m_nDigit;
public:
    Digit(int nDigit=0)
    {
        m_nDigit = nDigit;
    }

    Digit& operator++();
    Digit& operator--();

    int GetDigit() const { return m_nDigit; }
};

Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_nDigit == 9)
        m_nDigit = 0;
    // otherwise just increment to next number
    else
        ++m_nDigit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_nDigit == 0)
        m_nDigit = 9;
    // otherwise just decrement to next number
    else
        --m_nDigit;

    return *this;
}
```

كلاس Digit يقوم بإبقاء العدد بين 0 و 9 .ولقد قمنا بإعادة تعريف معامل الزيادة والإنقاص ++ - -- لكي يتعامل مع ال Digit ويحافظ علية ضمن المجال 0 و9 .

لو تلاحظ أن قمنا بإرجاع this* عند إعادة تعريف المعاملات جعلناها من نوع Digit وهي member function فا استخدمنا this* وهو عنصر من نوع Digit.

Overloading postfix increment and decrement

في العادة عند إعادة تعريف دوال من نفس الاسم لكن هناك اختلاف بالأرقام أو أنواع البيانات الممررة .وهذا هو الحاصل في إعادة تعريف

prefix and postfix جميعها تحمل نفس الشكل (operator++) وتأخذ بارمتر من نفس النوع (this*) كيف يتم إعادة تعريفها ؟

في السي ++ هناك طريقة لتفرقة بين النوعين وهي أخذ بارمتر من int

لكي نميز ال postfix عن ال prefix وهذا تعديل على مثال Digit

وفية الطريقتين ...

```
class Digit
{
private:
    int m_nDigit;
public:
    Digit(int nDigit=0)
    {
        m_nDigit = nDigit;
    }

    Digit& operator++(); // prefix
    Digit& operator--(); // prefix

    Digit operator++(int); // postfix
    Digit operator--(int); // postfix

    int GetDigit() const { return m_nDigit; }
};

Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_nDigit == 9)
        m_nDigit = 0;
    // otherwise just increment to next number
    else
        ++m_nDigit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_nDigit == 0)
        m_nDigit = 9;
    // otherwise just decrement to next number
    else
        --m_nDigit;

    return *this;
}
Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit cResult(m_nDigit);

    // Use prefix operator to increment this digit
    ++(*this);             // apply operator

    // return temporary result
    return cResult;       // return saved state
}
Digit Digit::operator--(int)
{
    // Create a temporary variable with our current digit
    Digit cResult(m_nDigit);

    // Use prefix operator to increment this digit
    --(*this);             // apply operator

    // return temporary result
    return cResult;       // return saved state
}
int main()
{
    Digit cDigit(5);
    ++cDigit; // calls Digit::operator++();
    cDigit++; // calls Digit::operator++(int);
```

هنا بعض النقاط المهم التي تحتاج إلى تركيز (يجب أن تعرف الفرق الأساسي بين postfix and prefix

1-قمنا بالتفرقة بين postfix وprefix بإضافة بارمتر من Int

2-البارمتر المرسل لم يحصل على اسم وهنا نخبر الكومبايلر أن هذا المتغير لا يستخدم

3- (وهو الأهم) postfix و prefixيؤدون نفس المهام كلها زادة الاوبجكت واحد ولكن الاختلاف في القيمة المرجعة .ال prefix يرجع الاوبجكت بعد عملية الزيادة بكل بساطة نحن نقوم بعملية الإنقاص والزيادة ثم نرجع this*

لكن ال postfix فيه اختلاف جوهري شوفوا الكود هذا قبل ما نشرح وكيف مخرجاته

```
Int main()

{

Int num=5;

Cout<<num++;

Cout<<num;

Return0;

}
```

في المرة الأولى راح يطبع 5 ولكن المرة الثانية راح يطبع 6

هنا الفكرة الجوهرية بال postfix أول شي تطبيق الدالة ثم القيام بعملية الزيادة .

نفس الفكرة نبي نسويها مع كلاس Digit نحتاج في البداية إلى إرجاع الاوبجكت قبل عملية الزيادة أو النقصان .

وهنا نقع في مشكلة

نحن لا نريد إرجاع الاوبجكت بعد عملية الزيادة أو النقصان وفي نفس الوقت لو أرجعنا الاوبجكت قبل العملية ما راح نجري عملية الزيادة أو النقصان !!!!!!!!؟؟؟!!.

فأضل طريقة لحل هذي المشكلة هي استخدام متغير مؤقت لحفظ قيمة الاوبجكت ثم زيادة الكلاس نفسه (استدعاء prefix) .

ثم إرجاع المتغير المؤقت إلى نقطة الاستدعاء وبكذا المستدعي حصل على نسخة من الاوبجكت قبل التعديل علية وبنفس الوقت قمنا بالتعديل على الاوبجكت نفسه .

لاحظ معي أن القيمة المرجعة من الاوفرلود ليست عنوان non-reference

لأن لا يمكن إرجاع عنوان إلى متغير .إنما هي قيمة وبذالك تمسح من الذاكرة عند إنتها الفنكشن function .

ولكي تفهموا النقطة الأخيرة شوفوا الكود هذا وجربوه ...

```cpp
#include <iostream>
using namespace std;


class Digit
{
private:
    int m_nDigit;
public:
    Digit(int nDigit=0)
    {
        m_nDigit = nDigit;
    }

    Digit& operator++(); // prefix
    Digit& operator--(); // prefix

    Digit operator++(int); // postfix
    Digit operator--(int); // postfix
        friend ostream &operator<<(ostream &out,Digit &digit);
};
ostream &operator<<(ostream &out,Digit &digit)
{
        out<<digit.m_nDigit<<endl;
        return out;
}
Digit& Digit::operator++()
{
    // If our number is already at 9, wrap around to 0
    if (m_nDigit == 9)
        m_nDigit = 0;
    // otherwise just increment to next number
    else
        ++m_nDigit;

    return *this;
}

Digit& Digit::operator--()
{
    // If our number is already at 0, wrap around to 9
    if (m_nDigit == 0)
        m_nDigit = 9;
    // otherwise just decrement to next number
    else
        --m_nDigit;

    return *this;
}

Digit Digit::operator++(int)
{
    // Create a temporary variable with our current digit
    Digit cResult(m_nDigit);

    // Use prefix operator to increment this digit
    ++(*this);             // apply operator

    // return temporary result
    return cResult;       // return saved state
}

Digit Digit::operator--(int)
{
    // Create a temporary variable with our current digit
    Digit cResult(m_nDigit);

    // Use prefix operator to increment this digit
    --(*this);             // apply operator

    // return temporary result
    return cResult;       // return saved state
}

int main()
{
    Digit cDigit(5),cDigit2(0);

    cout<<++cDigit; // calls Digit::operator++();


    cout<<cDigit2++; // calls Digit::operator++(int);

        cout<<cDigit2;

        system("pause");
        return 0;
}
```

6-Overloading the subscript operator

عند العمل مع المصفوفات فأننا نستخدم ([]) للوصل إلى أعضاء المصفوفة

```
anArray[0] = 7; // put the value 7 in the first element of
the array
```

نسوف الكلاس التالي IntList يحتوي على مصفوفة

```
class IntList
{
private:
    int m_anList[10];
};

int main()
{
    IntList cMyList;
    return 0;
}
```

وبمان المصفوفة من private فلا يمكن الوصول اليها مباشرة .

وبذالك نستخدم دوال ال set وget

```
class IntList
{
private:
    int m_anList[10];

public:
    void SetItem(int nIndex, int nData) { m_anList[nIndex] = nData;
}
    int GetItem(int nIndex) { return m_anList[nIndex]; }
};
int main()
{
    IntList cMyList;
    cMyList.SetItem(2, 3);

    return 0;
}
```

هنا قمنا بأسناد الرقم 3 الى العنصر رقم 2 او الرقم 2الى العنصر رقم 3

اذا لم نشاهد تعريف الدالة فالرؤيى غير واضحة .

فأفضل طريقة هي اعادة تعريف [] لكي نستطيع تعديل المصفوفة بدون اللجؤ الى دوال setو get. نشوف المثال التالي وهو اعادة تعريف [] subscript وهو من الانواع التي يجب ان تعرف كا member function

وتستقبل بارمتر واحد وهو من نوع انتجر وهو الاندكس ويرجع قيمته في المصفوفة

```
class IntList
{
private:
    int m_anList[10];

public:
    int& operator[] (const int nIndex);
};

int& IntList::operator[] (const int nIndex)
{
    return m_anList[nIndex];
}
```

الان عند استخدام [] مع الاوبجكت راح نستطيع الوصول الى m_anList

مباشرة والتعديل عليها لان عند استدعائها ترجع عنوان

```
IntList cMyList;
    cMyList[2] = 3; // set a value
    cout << cMyList[2]; // get a value

    return 0;
```

الكود هذا اوضح من الكود السابق بأستخدام set and get.

7-Overloading the parenthesis operator

المعامل الذي سوف نتكلم عنة هو الأقواس ( ) هذا النوع من المعامل يختلف اختلاف بسيط عن البقية وهو ان البقية محدد عدد البارمتر parameter

مثلا عملية == تحتوي على 2 parameter ثنائية وعملية ! تعتبر احادية لانها تستقبل بارمتر واحد (دائما) .

لكن عملية الاقواس يمكن ان تضع لها عدد لا نهائية من البارمتر على حسب prototype للدالة .

واعادة تعريف الاقواس لابد ان تكون الدالة كا member function .

نشوف المثال التالي

```
class Matrix
{
private:
    double adData[4][4];
public:
    Matrix()
    {
        // Set all elements of the matrix to 0.0
        for (int nCol=0; nCol<4; nCol++)
            for (int nRow=0; nRow<4; nRow++)
                adData[nRow][nCol] = 0.0;
    }
};
```

في درس اعادة تعريف الاقواس [] قومنا بأعادة تعريفها لكي نستطيع الوصول الى المصفوفة الاحادية (بُعد واحد) في اعضاء الكلاس الخاصة .

لا ن [] تحتوي على بارمتر واحد (one parameter) ولا نستطيع الى الوصول الى اكثر من بُعد .

لكن () يمكن ان تحتوي على الكثير من البارمتر ولذالك يعطين الصلاحيات في التحكم بالمصفوفة مهما كان بُعدها .

لو بغينا نعيد تعريف المثال السابق عندنا مصفوفة من بُعدين .

سوف نعيد تعريف () للوصول الى أي عنصر نريده في المصفوفة

نشوف المثال :

```
#include <cassert> // for assert()
class Matrix
{
private:
    double adData[4][4];
public:
    Matrix()
    {
        // Set all elements of the matrix to 0.0
        for (int nCol=0; nCol<4; nCol++)
            for (int nRow=0; nRow<4; nRow++)
                adData[nRow][nCol] = 0.0;
    }

    double& operator()(const int nCol, const int nRow);
};
```

الان يمكن تعريف Matrix والوصول الى الاعضاء بطريقة مباشرة

```
Matrix cMatrix;
cMatrix(1, 2) = 4.5;
std::cout << cMatrix(1, 2);

double& Matrix::operator()(const int nCol, const int nRow)
{
    assert(nCol >= 0 && nCol < 4);
    assert(nRow >= 0 && nRow < 4);

    return adData[nRow][nCol];
}
```

المخرجات راح تكون بالشكل التالي

```
4.5
```

الان خلونا نعيد تعريف () بدون بارمتر

```
#include <cassert> // for assert()
class Matrix
{
private:
    double adData[4][4];
public:
    Matrix()
    {
        // Set all elements of the matrix to 0.0
        for (int nCol=0; nCol<4; nCol++)
            for (int nRow=0; nRow<4; nRow++)
                adData[nRow][nCol] = 0.0;
    }

    double& operator()(const int nCol, const int nRow);
    void operator()();
};

double& Matrix::operator()(const int nCol, const int nRow)
{
    assert(nCol >= 0 && nCol < 4);
    assert(nRow >= 0 && nRow < 4);

    return adData[nRow][nCol];
}

void Matrix::operator()()
{
    // reset all elements of the matrix to 0.0
    for (int nCol=0; nCol<4; nCol++)
        for (int nRow=0; nRow<4; nRow++)
            adData[nRow][nCol] = 0.0;
}
```

واستخدامها يتم بالشكل التالي فقط تم اعادة تعريف () لمسح المصفوفة

```
Matrix cMatrix;
cMatrix(1, 2) = 4.5;
cMatrix(); // erase cMatrix
std::cout << cMatrix(1, 2);
```

المخرجات راح تكون

```
0
```

مثل ما شفنا () مرنة يمكن استخدامها بطرق كثيرة للقيام مهام عديدة .

هذا والله اعلم ...

8-Overloading typecasts

كما هو معروف بالسي ++ انة يمكن اسناد متغير من نوع int الى متغير آخر من نوع مثلا double بأستخدام ال typecast.

```
int nValue = 5;
double dValue = nValue; // int implicitly cast to a double
```

السي ++ عارفة كيف تحول بينهم معرفة مسبقا التحويل بين الانواع الرئيسية.

لكن لا تعرف كيف تتعامل مع أي كلاس ننشئه نحن فيجب تعريف طريقة لكي يمكن التحويل من نوع الى نوع آخر.

اعادة تعريف typecast تسمح لنا بتحويل الكلاس الى نوع آخر من البيانات.

نشوف الكلاس التالي:

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

الكلاس واضح وبسيط استخدمنا فية دالة set و get للوصول الى المتغير المحمي .

الان سوف نقوم بكتابة دالة تقوم بطباعة الكلاس Cents وهذا الدالة تستقبل متغير من نوع انتجر int فيجب علينا تحويل او بمعنى اصح ان نرسل قيمة تكون int الى الدالة. نشوف المثال لكي تتضح الامور :

```
void PrintInt(int nValue)
{
    cout << nValue;
}

int main()
{
    Cents cCents(7);
    PrintInt(cCents.GetCents()); // print 7

    return 0;
}
```

استخدمنا طريقة بسيطة لطباعة اذا كان البرنامج صغير فالطريقة هذي سهلة لكن اذا كان عندك برنامج كبير وعدد الفنكشن كثير او عدد البارمتر كثير فاراح يستهلك البرنامج اكثر من الازم.

افضل طريقة ان نعيد تعريف int cast لكي تحول من cents الى int

نشووف المثال :

```
class Cents
{
private:
    int m_nCents;
public:
    Cents(int nCents=0)
    {
        m_nCents = nCents;
    }

    // Overloaded int cast
    operator int() { return m_nCents; }

    int GetCents() { return m_nCents; }
    void SetCents(int nCents) { m_nCents = nCents; }
};
```

لاحظ معي هناك فراغ بين operator و int .

وفي مثالنا لو استدعينا الفانكشن PrintInt

راح نستدعيها بالشكل التالي:

```
int main()
{
    Cents cCents(7);
    PrintInt(cCents); // print 7

    return 0;
}
```

خلونا نفسر الي حصل ..:

- في البداية compiler راح يشوف ان PrintInt تستقبل بارمتر من نوع int.

- ثم يُلاحظ ان cCents ليس انتجر.

- ثم يقوم بالبحث اذا كان هناك طريقة خاصة لتحويل cCents الى int

طبعا راح يجدها ثم يستدعي دالة operator int ()

للقيام بهذي العملية والدالة هذي ترجع int ثم يرسل الى PrintInt.

ويمكن الان تحويل الكلاس Cents الى أي متغير انتجر مثل

```
Cents cCents(7);
int nCents = static_cast<int>(cCents);
```

ويمكن تحويل ان نوع من البيانات سواء التحويل بين انواع نحن انشأنها او مثل ماشفنا سابقا التحويل الى int .

خلونا نشوف مثال للتحويل بين انواع نحن ننشئها .

خلونا نكتب كلاس جديد اسمة دولار ثم نقوم باعادة تعريف Cents cast operator لكي نحول من السينت الى الدولار نشوف الكلاس ....

```
class Dollars
{
private:
    int m_nDollars;
public:
    Dollars(int nDollars=0)
    {
        m_nDollars = nDollars;
    }

     // Allow us to convert Dollars into Cents
     operator Cents() { return Cents(m_nDollars * 100); }
};
```

هذا يخولنا لتحويل أي اوبجكت من الكلاس دولار الى أي اوبجكت من الكلاس سينتس مباشرة ....

وداخل المين :

```
void PrintCents(Cents cCents)
{
    cout << cCents.GetCents();
}

int main()
{
    Dollars cDollars(9);
    PrintCents(cDollars); // cDollars will be cast to a Cents

    return 0;
}
```

المخرجات راح تكون

900

وذالك قمنا بالتحويل من دولار الى سنتس .

ومثل ما شفنا مرونتها يمكن استخدامها بطرقة كثيرة .

هذا والله اعلم ...